

Курс по Node.js

HTTP-сервер на Node.js

[Node.js v14.x]



На этом уроке

1. Узнаем, что такое http-сервер.
2. Выясним, чем GET-запрос отличается от POST-запроса.
3. Поговорим о том, как обработать эти запросы и как через Node.js показать html-страницу.
4. Узнаем, что такое кластеризация.
5. Разберём, как и зачем применяется стандартный модуль cluster в Node.js.
6. Подключим этот модуль в свою программу.
7. На практике посмотрим на одну из форм горизонтального масштабирования сервиса Node.js.

Оглавление

[Теория урока](#)

[Базовые понятия](#)

[HTTP-запросы](#)

[Модуль http](#)

[Объект request](#)

[Объект response](#)

[Маршрутизация](#)

[Доступ к параметрам GET-запроса](#)

[Доступ к данным POST-запроса](#)

[Выдача файлов](#)

[Масштабирование веб-сервера](#)

[Практическое задание](#)

[Глоссарий](#)

[Дополнительные материалы](#)

[Используемые источники](#)

Теория урока

Базовые понятия

Чтобы начать создавать свой первый сервер на Node.js, сначала повторим некоторые базовые понятия.

Http-сервер (веб-сервер) — сервер, принимающий HTTP-запросы и отвечающий на них с возможностью передачи HTML-страниц, изображений, файлов, медиапотоков и других данных. То есть, когда пользователь в браузере открывает страницу сайта, браузер (клиент) делает запрос к HTTP-серверу этого сайта. В ответ на свой запрос браузер получает HTML-страницу и показывает её пользователю.

HTTP — HyperText Transfer Protocol — протокол передачи гипертекста. Это протокол прикладного уровня (см. [сетевая модель OSI](#)) передачи данных. Протокол так называется, потому что изначально был придуман для передачи тех самых документов в формате HTML. Однако в настоящее время используется для передачи произвольных данных. На базе этого протокола построена большая часть интернета. Подробнее о HTTP-протоколе — в [этой статье](#).

Если вернуться к определению http-сервера, то становится понятно, почему его так называют.

Передача данных по HTTP обычно осуществляется через соединения TCP/IP. Так обычно называют набор различных протоколов, которые описывают способ передачи данных между различными устройствами в сети. Среди этого набора — протоколы TCP и IP, которые считаются основными.

TCP — Transmission Control Protocol — протокол управления передачей. Это протокол, предназначенный для управления передачей данных через сеть. Протокол включает в себя поток данных, предварительную установку соединения между двумя устройствами в сети, повторный запрос в случае потери данных, и устранение дублирования при получении данных. Таким образом, этот протокол обеспечивает целостность переданных данных.

IP — Internet protocol — межсетевой протокол. Это маршрутизируемый протокол сетевого уровня (см. [сетевая модель OSI](#)) стека TCP/IP. Протокол объединяет устройства сети в единую сеть и обеспечивает доставку пакетов данных между различными устройствами через произвольное количество промежуточных узлов, что называется маршрутизацией. TCP-протокол использует IP-протокол в качестве транспорта.

Чтобы осуществить запрос к http-серверу за какой-либо информацией, требуется указать URI.

URI — Uniform Resource Identifier — унифицированный идентификатор ресурса. Это последовательность символов, указывающая на путь до конкретного ресурса (документа, картинки, файла), над которым осуществляется операция. Например, на сайте GeekBrains есть обзорный вебинар по Node.js. Его URI выглядит так: <https://geekbrains.ru/events/2>.

HTTP-запросы

Чтобы осуществить запрос, для веб-сервера требуется указать не только URI, но и метод запроса.

Метод http-запроса — это последовательность из любых символов, кроме управляющих и разделителей. Такая последовательность указывает на основную операцию над ресурсом. Чаще всего название метода — это короткое английское слово, написанное заглавными буквами.

Есть несколько общепринятых методов, относящихся к http-запросам. Наиболее часто используются следующие:

1. GET чаще всего используется для получения данных указанного ресурса. Например, GET-запрос HTML-страницы по определённому URI вернёт эту страницу. В GET-запросе можно передавать параметры выполнения запроса, для этого надо добавить их в URI запроса после символа «?»:

“/path/resource?param1=value1¶m2=value2”

2. POST используется для передачи данных ресурсу. Например, если на сайте пользователь оставляет комментарий к статье — это POST-запрос. В таком запросе параметры, например, содержание комментария и идентификатор пользователя, который его оставил, передаются в теле запроса. Метод POST также используется для загрузки файлов и картинок на сервер.
3. PUT во многом похож на метод POST, но отличается от него фундаментальным предназначением. Метод POST предполагает обработку передаваемых данных, а PUT — что передаваемые данные уже соответствуют ресурсу.
4. DELETE используется для удаления ресурса с соответствующим URI. Например, при удалении комментария к статье или при удалении самой статьи.

Важно!

Использование тех или иных методов при построении клиент-серверного взаимодействия строго не регламентировано и во многом зависит от программистов. Однако есть некий набор общепринятых практик (best practices), которых рекомендуется придерживаться. Это облегчает взаимодействие других программистов с предоставленным кодом, сервером, приложением. Однако нет какого-то конкретного сборника best practices. Узнать, что лучше всего подходит для того или иного случая, можно, только постоянно изучая тематические форумы и статьи, общаясь с другими программистами, а также регулярно практикуясь и развиваясь. Пример такой статьи — [здесь](#).

В запросы включаются следующие элементы:

1. Метод запроса.
2. Путь к ресурсу.
3. Версия HTTP-протокола.
4. Заголовки (Headers) — предоставляют дополнительную информацию для сервера. они представлены в виде «ключ:значение». Заголовки бывают следующих видов:
 - а. Общие заголовки — применяются как к запросам, так и к ответам, не имеют отношения к данным, которые передаются в теле.
 - б. Заголовки запроса — содержат дополнительную информацию о запрашиваемом ресурсе или о клиенте, который эту информацию запрашивает.

- c. Заголовки ответа — содержат дополнительную информацию об ответе или о сервере, который предоставил ответ.
 - d. Заголовки сущностей — содержат информацию о теле запроса, например, о формате передаваемых данных, длине содержимого и т. д.
5. Тело запроса — здесь передаются данные, например, в POST-запросе.

Модуль http

Для создания и конфигурирования http-серверов в Node.js есть стандартный модуль http. Создадим наш первый http-сервер, который будет возвращать Hello World!. Для этого нам понадобится несколько строк кода, который мы запишем в файл server.js:

```
const http = require('http');

http.createServer((request, response) => {
  response.end('Hello World!');
}).listen(3000, 'localhost');
```

Теперь разберём, что здесь происходит.

1. Инициализация модуля http. Это стандартный модуль, он не требует дополнительной установки через npm.
2. Метод `createServer` модуля http создаёт экземпляр класса `http.Server`. Этот класс предоставляет различные методы для конфигурирования http-сервера.
3. Метод `listen(port, host)` запускает сервер для прослушивания соединений. После вызова этого метода сервер начинает прослушивать соединения с соответствующим хостом и портом. В нашем случае — это порт 3000 и localhost в качестве хоста, так как сервер слушает запросы на локальной машине.
4. Объект класса `http.Server`, созданный нами, имеет свои события и позволяет создавать для них обработчики. Событиями могут быть:
 - входящий запрос к серверу (событие `request`);
 - закрытие сервера (событие `close`);
 - новое TCP или IP-соединение (событие `connection`);
 - прочие.

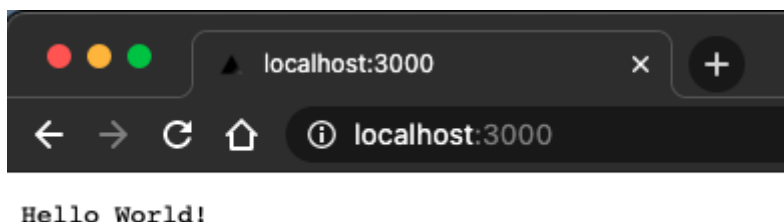
В метод создания объекта сервера передаётся функция `requestListener`. Она автоматически станет обработчиком события `request`, то есть будет срабатывать каждый раз, когда приходит новый запрос на сервер. Эта функция принимает на вход два аргумента — объект запроса

request и объект ответа response. Вызов метода response.end() с аргументом в виде строки Hello World! и будет ответом сервера на любой запрос к нему.

Теперь запустим сервер и создадим к нему запрос. Запуск осуществляем уже привычной командой:

```
node server.js
```

Далее сделаем запрос к серверу. Для выполнения простого GET-запроса откроем [этот адрес](#) в браузере. Результат будет следующим:



Рассмотрим подробнее объект request и response.

Объект request

Это объект запроса к серверу. Он создается http-сервером и передается первым аргументом в функцию-обработчик события request. По сути, это расширенный объект потока на чтение. Такой объект хранит в себе всю информацию о запросе, включая переданные данные и заголовки запроса. Из него считываются передаваемые данные, если работать с ним как с потоком. Часто используемые свойства этого объекта:

1. headers — объект, который содержит заголовки запроса.
2. method — метод запроса.
3. url — строка, содержащая запрашиваемый url.

Для закрепления материала выведем данные по запросу в терминал.

```
const http = require('http');

http.createServer((request, response) => {
  console.log(`Url запроса: ${request.url}`);
  console.log(`Метод запроса: ${request.method}`);
  console.log(`Заголовки запроса: \n${JSON.stringify(request.headers)}\n`);

  response.end('Hello World!');
}).listen(3000, 'localhost');
```

В терминале мы получаем нечто подобное:

```
Url запроса: /
Метод запроса: GET
Заголовки запроса:
{"host":"localhost:3000","connection":"keep-alive","cache-control":"max-age=0","
sec-ch-ua":"\"Chromium\";v=\"88\"\", \"Google Chrome\";v=\"88\"\", \";Not A
Brand\";v=\"99\"\"\", \"sec-ch-ua-mobile\":\"?0\", \"upgrade-insecure-requests\":\"1\", \"user-
agent\":\"Mozilla/5.0 (Macintosh; Intel Mac OS X 11_1_0) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/88.0.4324.150
Safari/537.36\", \"accept\":\"text/html,application/xhtml+xml,application/xml;q=0.9,i
mage/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
\", \"sec-fetch-site\":\"none\", \"sec-fetch-mode\":\"navigate\", \"sec-fetch-user\":\"?1\", \"sec
-fetch-dest\":\"document\", \"accept-encoding\":\"gzip, deflate,
br\", \"accept-language\":\"ru-RU, ru;q=0.9, en-US;q=0.8, en;q=0.7\"}

Url запроса: /favicon.ico
Метод запроса: GET
Заголовки запроса:
{"host":"localhost:3000","connection":"keep-alive","pragma":"no-cache","cache-co
ntrol":"no-cache","sec-ch-ua":"\"Chromium\";v=\"88\"\", \"Google
Chrome\";v=\"88\"\", \";Not A
Brand\";v=\"99\"\"\", \"sec-ch-ua-mobile\":\"?0\", \"user-agent\":\"Mozilla/5.0 (Macintosh;
Intel Mac OS X 11_1_0) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/88.0.4324.150
Safari/537.36\", \"accept\":\"image/avif,image/webp,image/apng,image/svg+xml,image/*
,*/*;q=0.8\", \"sec-fetch-site\":\"same-origin\", \"sec-fetch-mode\":\"no-cors\", \"sec-fetch-
dest\":\"image\", \"referrer\":\"http://localhost:3000/\", \"accept-encoding\":\"gzip,
deflate, br\", \"accept-language\":\"ru-RU, ru;q=0.9, en-US;q=0.8, en;q=0.7\"}
```

В браузере Google Chrome помимо GET-запроса страницы также происходит GET-запрос файла favicon.ico. Далее мы рассмотрим разные методы обработки запросов.

Объект response

Это объект ответа на запрос. Представляет собой расширенный объект потока на запись. Он создается http-сервером, а не вручную программистом, и передается вторым аргументом в функцию-обработчик события request. Часто используемые методы этого объекта:

1. `setHeader(name, value)` используется для добавления заголовка ответа. Здесь `name` — это название заголовка строкой, `value` — его значение любого типа. Однако, если тип значения `value` будет отличаться от строки, такое значение всё равно будет конвертироваться в строку во время передачи данных по сети. Обычно вызывается несколько раз для добавления различных заголовков.

2. `writeHead(statusCode[, statusCode], statusMessage[, statusMessage], headers)` отправляет заголовки ответа на запрос. Параметр `statusCode` — это статус ответа из трёх цифр, например, 404. Подробнее о статусах HTTP-ответов — [здесь](#).

Параметр `statusMessage` считается опциональным. Иногда в него передаётся строковое сообщение. Параметр `headers` может быть объектом или массивом. Если это объект, то ключами считаются названия заголовков. Когда это массив, то элементы с чётными индексами представляют собой названия заголовков, а с нечётными — значения. Этот метод вызывается только один раз, до вызова метода `end`.

3. Если не использовать метод `writeHead`, то статус ответа и сообщение устанавливаются через поля `statusCode` и `statusMessage`, указав им соответствующие значения.

4. `write(chunk[, encoding][, callback])` отправляет части данных, которые требуется передать в теле ответа. Такой метод вызывается несколько раз, пока все данные не будут переданы.

Параметр `chunk` может быть строкой или экземпляром класса `Buffer`. При первом вызове отправляются буферизированные заголовки, например, если они не отправлены методом `writeHead`. С заголовками также посылается переданная часть данных. При повторных вызовах метод отправляет только переданные ему данные.

5. `end([data[, encoding]][, callback])` сообщает серверу, что все заголовки и данные ответа отправлены. Сервер, в свою очередь, фиксирует ответ завершённым.

Метод `end` вызывается в конце каждого запроса. Если параметры `data` и `encoding` переданы, то поведение метода будет аналогично вызову метода `write(data, encoding)` и следующему за ним вызову `end(callback)`. После передачи параметра `callback` функция коллбэк будет вызвана после того, как поток ответа завершится.

Разберём это на практике. Сформируем простой вывод Hello World! через передачу заголовков, кода ответа и передачу фразы частями по словам.

```
http = require('http');

http.createServer((request, response) => {
  response.setHeader('Content-Type', 'text/html');
  response.setHeader('Custom-header-1', 'Custom header value 1');

  response.writeHead(200, {
    'Custom-header-2': 'Custom header value 2',
  });

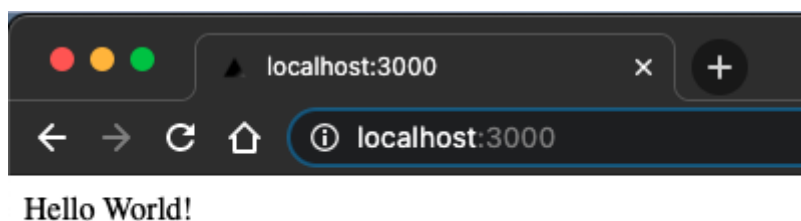
  response.write('Hello');
  response.write(' ');
  response.write('World');

  response.end('!');
```

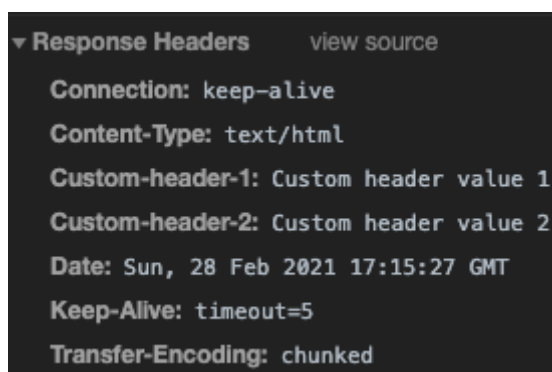


```
}).listen(3000, 'localhost');
```

Здесь мы использовали методы, разобранные ранее. Если выполнить запрос из браузера, то можно увидеть, что все данные успешно приходят, и суть сообщения Hello World! не изменилась. Внешне изменился только шрифт сообщения. Это происходит, потому что ранее по умолчанию контент ответа воспринимался браузером как text/plain. Сейчас же, через установку заголовка Content-Type мы явно сказали браузеру, что ответ приходит в формате text/html, и браузер отрисовал его по-другому.



Если мы откроем в браузере инструменты разработчика и посмотрим на заголовки ответа, то увидим, что все заголовки переданы.



Сейчас в примере все запросы обрабатываются одинаково. Однако мы помним, что для различных операций предусмотрены разные методы запросов. Научимся с ними работать. Для этого создадим некие правила маршрутизации — опишем некие запросы и то, как они будут обрабатываться.

Маршрутизация

Для начала научимся отделять GET-запросы от POST-запросов, так как это два наиболее часто встречающихся метода. Научившись разделять запросы по этим двум методам, мы поймём, как это делать и для других методов.

Вернёмся к началу. У нас есть сервер, который возвращает сообщение Hello World! в ответ на любой запрос:

```
const http = require('http');
```

```
http.createServer((request, response) => {
  response.end('Hello World!');
}).listen(3000, 'localhost');
```

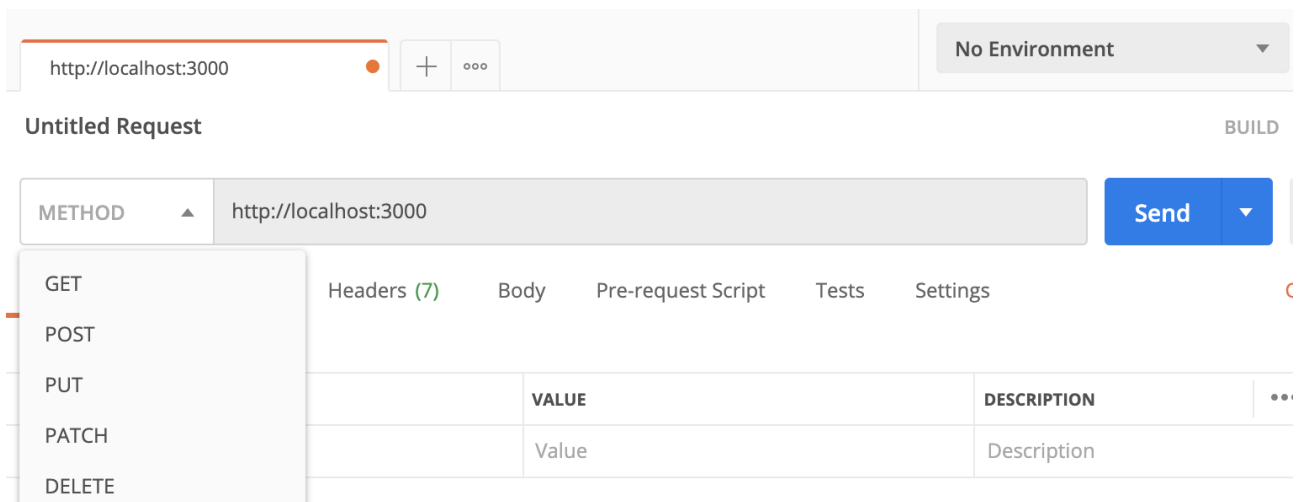
Сделаем так, чтобы он возвращал Hello World! только в ответ на GET-запрос, а на все остальные отвечал Method Not Allowed. Для этого вспомним, что в объекте запроса есть поле method.

```
const http = require('http');

http.createServer((request, response) => {
  if (request.method === 'GET') {
    response.end('Hello World!');
  } else {
    response.end('Method Not Allowed');
  }
}).listen(3000, 'localhost');
```

Важно!

Чтобы делать запросы к серверу разными методами, очень удобно использовать программу Postman. На [официальном сайте](#) можно найти и скачать бесплатную версию, а также воспользоваться веб-версией. Для личной разработки бесплатная версия обладает полным инструментарием. Интерфейс у программы очень простой и интуитивно понятный (см. [скриншот ниже](#)).



Теперь, если мы сделаем любой запрос, у которого метод не GET, то получим сообщение Method Not Allowed.

Сообщение Method Now Allowed выбрано неслучайно. Это устоявшийся ответ сервера, когда запрос на операцию с ресурсом приходит с таким методом, который не поддерживается для этого ресурса. Кроме сообщения, этот ответ включает в себя также код статуса ответа 405. Добавим его.

```
response.statusCode = 405;
response.end('Method Not Allowed');
```

Доступ к параметрам GET-запроса

Мы уже знаем, что в GET-запросах параметры передаются в строке адреса ресурса. Они также разделяются знаком «?» при добавлении параметров к URI ресурса и знаком «&» при разделении самих параметров.

Так выглядят URI нашего GET-запроса с параметрами:

```
http://localhost:3000?param1=value1&param2=value2
```

Чтобы получить доступ к этим параметрам на сервере, воспользуемся полем url объекта запроса request и методом parse модуля стандартного модуля url:

```
const queryParams = url.parse(request.url, true).query;
```

В этом примере мы передаём в метод parse url запроса, и вторым параметром передаём true — это значит, что метод распарсит строку запроса и превратит её в объект, доступный по ключу query. Выведем это в терминал:

```
[Object: null prototype] { param1: 'value1', param2: 'value2' }
```

Доступ к данным POST-запроса

Входящий запрос для http-сервера — это поток Readable. Для чтения переданных данных нам понадобятся знания из третьего урока.

Создадим переменную для POST-запросов. В неё мы будем складывать данные, полученные из потока:

```
} else if (request.method === 'POST') {
  let data = '';
}
```

Чтобы считывать данные из потока на чтение, воспользуемся уже знакомой концепцией обработчика события 'data':

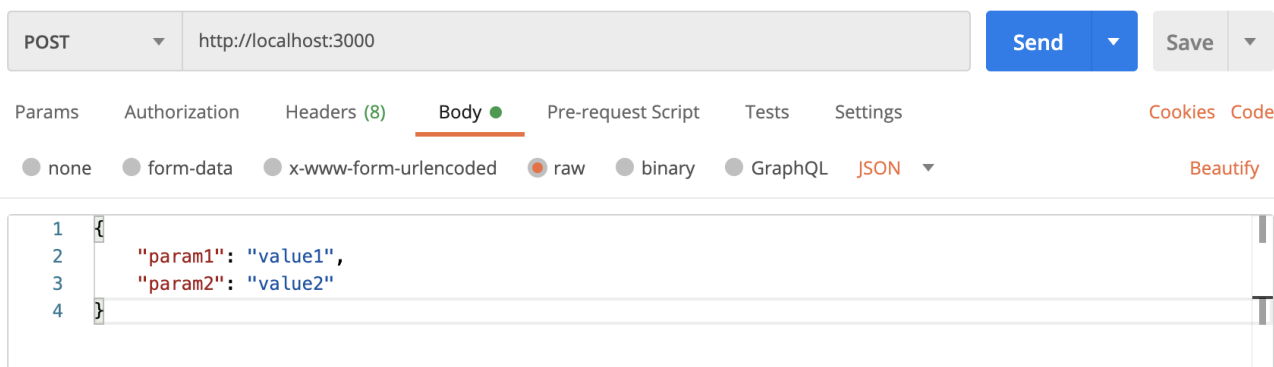
```
request.on('data', chunk => {  
  data += chunk;  
});
```

По окончании приёма данных отправим считанные данные обратно клиенту. Таким образом, мы убедимся, что сервер принял их и обработал.

Чтобы поймать момент окончания чтения данных, относящихся к потоку запроса, создадим обработчик события 'end':


```
request.on('end', () => {  
  console.log(data);  
  response.end(data);  
});
```

Далее отправляем POST-запрос с JSON-данными, чтобы проверить, что получилось. Такой POST-запрос в Postman будет выглядеть следующим образом:



Ответ от сервера, полученный в Postman:

Body Cookies Headers (4) Test Results

Pretty Raw Preview Visualize Text 

```
1 {
2   "param1": "value1",
3   "param2": "value2"
4 }
```

И в терминале:

```
{
  "param1": "value1",
  "param2": "value2"
}
```


Сервер принял данные, которые мы передали ему в POST-запросе. Однако в ответ на запрос они пришли в текстовом формате, о чём свидетельствует надпись Text в окне ответа.


Однако отправлять JSON в формате Text — плохая практика.

Переведём наш контент в JSON-формат. Для этого воспользуемся заголовком Content-Type:

```
response.writeHead(200, { 'Content-Type': 'json' });
```

Выполним еще один запрос и посмотрим на ответ:

Body Cookies Headers (5) Test Results  200 OK

Pretty Raw Preview Visualize JSON 

```
1 {
2   "param1": "value1",
3   "param2": "value2"
4 }
```

Чтобы в коде получить доступ к данным, распарсим пришедший JSON, используя функцию JSON.parse:

```
const parsedData = JSON.parse(data);
```

```
console.log(parsedData);
```

Результат в терминале:

```
{ param1: 'value1', param2: 'value2' }
```

Мы научились работать с основными запросами — GET и POST, а также обрабатывать параметры и данные, передающиеся с этими запросами.

Выдача файлов

Когда пользователь открывает сайт, и происходит первый запрос к http-серверу, задача сервера — вернуть html-файл, в котором содержатся все требуемые скрипты и файлы стилей.

Теперь переделаем в нашем сервере часть, которая обрабатывает GET-запросы, и научим сервер возвращать html-файл.

Создадим файл index.html:

```
<html>
  The first html-page
</html>
```

Важно!

Имя index выбрано неслучайно. Это традиционное название html-файла, который используется в качестве точки входа в сайт. Если в URL указывается не конкретный файл, а каталог — то веб-сервер по умолчанию будет искать файл с названием index.html.

Чтобы отдать файл в ответ на запрос, используя веб-сервер, сначала надо этот файл прочитать. Так как ответ на запрос — это поток на запись, создадим поток для чтения файла и направим его в поток ответа на запрос, используя метод pipe() для связи:

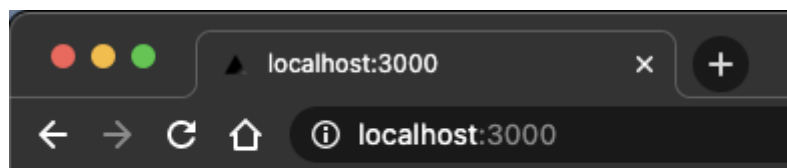
```
// Создаем строку, описывающую путь к файлу
const filePath = path.join(__dirname, 'index.html');

// Создаем объект потока на чтение файла
readStream = fs.createReadStream(filePath);

// В заголовке указываем тип контента html
response.writeHead(200, { 'Content-Type': 'text/html' });

// Направляем потока на чтение файла в потока на запись (поток ответа)
readStream.pipe(response);
```

Сделаем запрос из браузера:



The first html-page

Аналогичным образом раздаются так называемые статические файлы. Это такие файлы, которые не содержат рендеримый контент: файлы, PDF, изображения и прочее.

Масштабирование веб-сервера

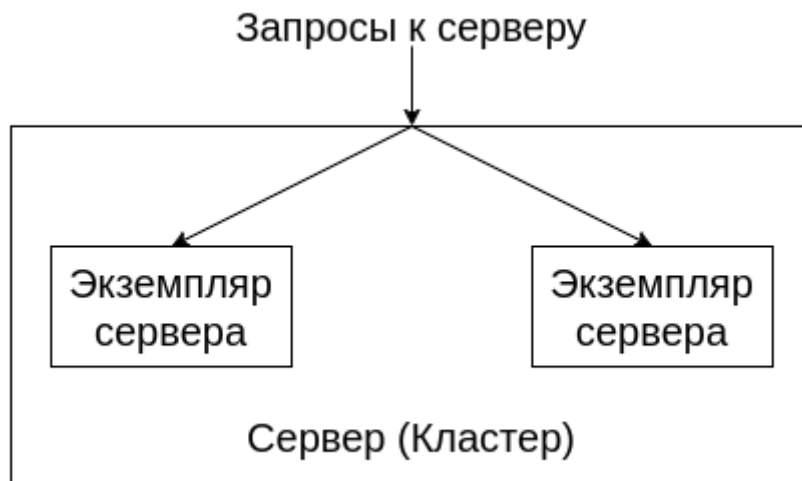
Каждый сервер имеет свою ограниченную пропускную способность. То есть количество запросов, которые он обрабатывает за единицу времени, так как время конечно. Пока увеличивается востребованность веб-сервиса, количество запросов к его серверу также растёт. Таким образом, рано или поздно перед программистами, которые его разрабатывают, встанет задача по масштабированию своих ресурсов.

Масштабируемость — это способность системы справляться с увеличением рабочей нагрузки при добавлении ресурсов.

Есть два подхода к масштабированию — вертикальное и горизонтальное.

Вертикальное масштабирование — это подход, когда производительность сервера увеличивается путём увеличения его физических ресурсов: более мощный CPU, увеличение количества оперативной памяти и т. д.

Горизонтальное масштабирование — это подход, когда для увеличения производительности сервера увеличивается количество его экземпляров (инстансов). Например, если запустить два одинаковых сервера, то суммарно они обработают в два раза больше запросов, чем один сервер. Каждый такой сервер называют не сервером, а экземпляром сервера или инстансом сервера. Сервер, в этом случае, это совокупность запущенных инстансов. Такую совокупность ещё называют кластером.



Так как Node.js — это однопоточная среда, то, соответственно, один Node.js-процесс не может по максимуму задействовать все доступные на физической машине мощности процессора. В общем случае задействовано будет только одно ядро процессора.

Для увеличения пропускной способности веб-сервера в Node.js есть стандартный модуль `cluster`. Этот модуль даёт возможность основному процессу запустить несколько дополнительных независимых экземпляров приложения, которые будут прослушивать один и тот же порт. Все входящие запросы распределятся между этими экземплярами. Совокупность основного процесса и дочерних экземпляров и будет кластером.

Модуль `cluster` поддерживает два способа распределения входящих запросов между дочерними процессами.

1. Циклическое распределение или распределение по алгоритму `round-robin`. В таком случае главный процесс (`master`-процесс) прослушивает все входящие соединения и распределяет их между дочерними процессами, воркерами, по алгоритму `round-robin`.

Суть этого алгоритма состоит в круговом цикле, где перебираются элементы распределённой системы. В нашем случае — это воркеры. Подробнее об этом алгоритме — [здесь](#). Такой способ используется по умолчанию во всех системах, кроме Windows.

2. Второй способ заключается в создании `master`-процессом прослушивающего сокета и отправки его воркерам. Таким образом, воркеры принимают входящие соединения напрямую. Подробнее о сокетах мы поговорим в следующем уроке, сейчас только отметим, что это способ коммуникации в режиме реального времени между двумя и более узлами.

Отличие от обычных запросов здесь в том, что соединение между узлами поддерживается неразрывным в течение всего времени, пока идёт обмен данными между узлами. Каждый

запрос — это отдельное TCP-соединение, которое открывается при выполнении запроса и закрывается после получения ответа.

Для достижения максимальной производительности не рекомендуется запускать процессов больше, чем есть ядер у процессора.

Теперь, когда разобрались немного с теорией, научим наш сервер запускать дополнительные экземпляры и посмотрим на практике, как это работает.

Для начала узнаем, сколько ядер процессора на машине для запуска сервера. От этого будет зависеть количество запускаемых воркеров. Чтобы узнать эту информацию, воспользуемся стандартным модулем [os](#).

```
const os = require('os');
const numCPUs = os.cpus().length;
```

Далее подключаем модуль cluster и учимся определять, в каком процессе мы находимся — в главном или в одном из дочерних. Это легко сделать, используя следующую конструкцию:

```
const cluster = require('cluster');

if (cluster.isMaster) {
  // Здесь будет код для главного процесса
} else {
  // А здесь будет код для всех дочерних процессов
}
```

Для главного процесса значение поля isMaster модуля cluster будет равно true. По аналогии с этим полем есть также поле isWorker. Оно равняется true для любого из дочерних процессов.

Определить в главном процессе, что мы находимся в одном из воркеров, можно по переменной окружения process.env.NODE_UNIQUE_ID. Если она undefined, значит, процесс главный. На основе этого и определяются значения полей isMaster и isWorker.

В главном процессе требуется создать дочерние. Для этого в модуле cluster есть метод fork([env]). Он создаёт новый рабочий процесс, при необходимости в него можно передать объект с переменными окружения env. Метод fork, как правило, вызывается только из главного процесса.

```
if (cluster.isMaster) {
  console.log(`Master ${process.pid} is running`);

  for (let i = 0; i < numCPUs; i++) {
    console.log(`Forking process number ${i}...`);
    cluster.fork();
  }
}
```

```
} else {  
  // А здесь будет код для всех дочерних процессов  
}
```

Если же процесс принадлежит воркеру, то потребуется запустить http-сервер, созданный ранее.

Итоговый код получается следующим:

```
const cluster = require('cluster');  
const os = require('os');  
const http = require('http');  
const fs = require('fs');  
const path = require('path');  
  
const numCPUs = os.cpus().length;  
  
if (cluster.isMaster) {  
  console.log(`Master ${process.pid} is running`);  
  
  for (let i = 0; i < numCPUs; i++) {  
    console.log(`Forking process number ${i}...`);  
    cluster.fork();  
  }  
} else {  
  console.log(`Worker ${process.pid} started...`);  
  
  http.createServer((request, response) => {  
    console.log(`Worker ${process.pid} handle this request...`);  
  
    setTimeout(() => {  
      if (request.method === 'GET') {  
        // Создаем строку, описывающую путь к файлу  
        const filePath = path.join(__dirname, 'index.html');  
  
        // Создаем объект потока на чтение файла  
        readStream = fs.createReadStream(filePath);  
  
        // В заголовке указываем тип контента html  
        response.writeHead(200, { 'Content-Type': 'text/html' });  
  
        // Направляем потока на чтение файла в потока на запись (поток ответа)  
        readStream.pipe(response);  
      } else if (request.method === 'POST') {  
        let data = '';  
  
        request.on('data', chunk => {  
          data += chunk;  
        });  
  
        request.on('end', () => {  
          const parsedData = JSON.parse(data);  
        });  
      }  
    }, 1000);  
  });  
}
```

```
    console.log(parsedData);

    response.writeHead(200, { 'Content-Type': 'json' });
    response.end(data);
  });
} else {
  response.statusCode = 405;
  response.end();
}
}, 5000);
}).listen(3000, 'localhost');
}
```

Если внимательно изучить этот код, то можно заметить, что там, где должен быть код воркера — находится веб-сервер, который мы написали ранее. Однако внутри этого сервера обработка входящих запросов обернута в метод `setTimeout` с задержкой 5 секунд. Это сделано исключительно с одной целью — на практике увидеть, что запросы обрабатываются разными воркерами. Так как без `setTimeout` к тому моменту, когда мы вручную пошлём следующий запрос через Postman, воркер уже освободится от выполнения предыдущего и возьмёт следующий запрос. С искусственной задержкой мы можем послать два запроса из Postman и увидеть в терминале, что их обрабатывают разные воркеры.

В таком случае в терминале мы увидим нечто подобное:

```
Master 34235 is running
Forking process number 0...
Forking process number 1...
Forking process number 2...
Forking process number 3...
Forking process number 4...
Forking process number 5...
Forking process number 6...
Forking process number 7...
Worker 34242 started...
Worker 34249 started...
Worker 34261 started...
Worker 34243 started...
Worker 34250 started...
Worker 34268 started...
Worker 34282 started...
Worker 34274 started...
Worker 34242 handle this request...
Worker 34249 handle this request...
```

Мы рассмотрели базовый инструментарий модуля `cluster` без глубокого погружения. Теперь у нас есть представление о возможностях этого модуля. С полученными знаниями нам не составит труда

разобраться, например, с передачей сообщений из master-процесса в воркеры и обратно, так как этот процесс основывается на событиях, с которыми мы уже знакомы.

Практическое задание

Используя наработки практического задания прошлого урока, создайте веб-версию приложения. Сделайте так, чтобы при запуске она:

- показывала содержимое текущей директории;
- давала возможность навигации по каталогам из исходной папки;
- при выборе файла показывала его содержимое.

Глоссарий

1. **Горизонтальное масштабирование** — это подход, когда для увеличения производительности сервера увеличивается количество его экземпляров (инстансов). Например, если запустить два одинаковых сервера, то суммарно они обработают в два раза больше запросов, чем один сервер. Каждый такой сервер называют не сервером, а экземпляром сервера или инстансом сервера. Сервером в этом случае становится совокупность запущенных инстансов. Такую совокупность ещё называют кластером.
2. **Масштабируемость** — способность системы справляться с увеличением рабочей нагрузки при добавлении ресурсов.
3. **Протокол TCP** — Transmission Control Protocol — протокол управления передачей. Это протокол, предназначенный для управления передачей данных через сеть. Протокол включает в себя поток данных, предварительную установку соединения между двумя устройствами в сети, повторный запрос в случае потери данных и устранение дублирования при получении данных. Таким образом, этот протокол обеспечивает целостность переданных данных.
4. **index.html** — это традиционное название html файла, который используется в качестве точки входа в сайт. Если в URL указывается не конкретный файл, а каталог — веб-сервер будет искать по умолчанию файл с названием index.html.
5. **IP** — Internet protocol — межсетевой протокол. Это маршрутизируемый протокол сетевого уровня (см. [сетевая модель OSI](#)) стека TCP/IP. Такой протокол объединяет устройства сети в единую сеть и обеспечивает доставку пакетов данных между различными устройствами через произвольное количество промежуточных узлов (маршрутизация). Протокол TCP использует протокол IP в качестве транспорта.
6. **HTTP** — HyperText Transfer Protocol — протокол передачи гипертекста. Это протокол прикладного уровня передачи данных. Изначально он появился для передачи документов в формате HTML, но в настоящее время используется для передачи произвольных данных.

7. **Http-сервер** (веб-сервер) — это сервер, принимающий HTTP-запросы и отвечающий на них, с возможностью передачи HTML-страниц, изображений, файлов, медиапотоков и других данных.
8. **Round-robin-алгоритм** (round-robin — циклический) — это алгоритм распределения нагрузки распределённой вычислительной системы методом перебора и упорядочения её элементов по круговому циклу.
9. **TCP/IP** — это набор различных протоколов, которые описывают способ передачи данных между различными устройствами в сети. TCP и IP — два основных протокола.
10. **URI** — Uniform Resource Identifier — унифицированный идентификатор ресурса. Это последовательность символов, указывающая на путь к конкретному ресурсу (документа, картинки, файла), над которым требуется осуществить операцию.

Дополнительные материалы

1. [Официальная документация.](#)
2. Статья [«Сетевая модель OSI».](#)
3. Статья [«HTTP-протокол».](#)
4. Статья [Transmission Control Protocol.](#)
5. Статья [«Протокол IP».](#)
6. Статья [URI.](#)
7. Статья [REST API Best Practices.](#)
8. Статья [«Коды ответа HTTP».](#)
9. Статья [Index.html.](#)
10. Статья [«Алгоритм Round-robin».](#)
11. [Модуль OS.](#)

Используемые источники

1. [Официальная документация.](#)
2. Статья [HTTP.](#)
3. Статья [«Обзор HTTP».](#)

4. Статья [Transmission Control Protocol](#).
5. Статья [«Протокол IP»](#).
6. Статья [«Масштабируемость»](#).